# Collecting Performance Data with PAPI-C


Dan Terpstra[1], Heike Jagode[1], Haihang You[3] Jack Dongarra[1,2]

[1] The University of Tennessee
[2] Oak Ridge National Laboratory
[3] National Institute for Computational Sciences
{terpstra,jagode,you,dongarra}@eecs.utk.edu



**Abstract.** Modern high performance computer systems continue to increase in size and complexity. Tools to measure application performance in these increasingly complex environments must also increase the richness of their measurements to provide insights into the increasingly intricate ways in which software and hardware interact. PAPI (the Performance API) has provided consistent platform and operating system independent access to CPU hardware performance counters for nearly a decade. Recent trends toward massively parallel multi-core systems with often heterogeneous architectures present new challenges for the measurement of hardware performance information, which is now available not only on the CPU core itself, but scattered across the chip and system. We discuss the evolution of PAPI into Component PAPI, or PAPI-C, in which multiple sources of performance data can be measured simultaneously via a common software interface. Several examples of components and component data measurements are discussed. We explore the challenges to hardware performance measurement in existing multi-core architectures. We conclude with an exploration of future directions for the PAPI interface.



**Key words:** Hardware Performance Counters, Performance Analysis, Performance Tools, Multicore, System Health


## 1 Introduction

The use of hardware counters to measure and improve software performance has become an accepted and integral method in the software development cycle [1]. Hardware counters, which are usually implemented as a small set of registers onto which can be mapped a larger set of performance related events, can provide accurate and detailed information on a wide range of hardware performance metrics. PAPI, the Performance Application Programming Interface, provides an easy to use, common API to application and tool developers to supply them with the information they may need to analyze, model and tune their software on a wide range of different platforms.

In addition to the counters found on CPUs, a large amount of hardware monitoring information is also available in other sub-sytems throughout modern computer architectures. Many network switches and network interface cards

(NICs) contain counters that can monitor various events related to performance and reliability. Possible events include checksum errors, dropped packets, and packets sent and received. Although the set of network events is necessarily somewhat dependent on the underlying hardware, extending PAPI to the network monitoring domain can provide a portable way to access native network events and allow correlation of network events with other domains. Because communication in OS-bypass networks such as Myrinet and Infiniband is handled asynchronously to the application, hardware monitoring, in addition to being low overhead, may be the only way to obtain some important data about communication performance.

As processor densities climb, the thermal properties and energy usage of high performance systems are becoming increasingly important. Such systems contain large numbers of densely packed processors which require a great deal of electricity. Power and thermal management issues are becoming critical to successful resource utilization [2, 3]. Standardized interfaces for accessing the thermal sensors are available, but may be difficult to use for runtime power-performance adaptation [4]. Extending the PAPI interface to simultaneously monitor processor metrics and thermal sensors can provide clues for correlating algorithmic activity with thermal system responses thus help in developing appropriate workload distribution strategies. We show the results of using the extended version of PAPI to simultaneously monitor processor counters, ACPI thermal sensors, and Myrinet network counters while running the FFTE and HPL HPC Challenge benchmarks [12] on a AMD Opteron Linux cluster.

Modifying and extending a library with a broad user base such as PAPI requires care to preserve simplicity and backward compatibility as much as possible while providing clean and intuitive access to important new capabilities. We discuss modifications to PAPI to provide support for the simultaneous measurement of data from multiple counter domains.

With the advent of multi-core processors and the inexorable increase in core counts per chip, interactions between cores and contention for shared resources such as last level caches or memory bus bandwidth become increasingly important sources of potential performance bottlenecks. Individual vendors have chosen different paths to provide access to hardware performance monitoring for these shared resources, each with their own problems and issues. We explore some of these approaches and their implications for performance measurement, and provide an example measurement of cache data on a real application in a multi-core environment to illustrate these issues.

## 2 Extending PAPI to Multiple Measurement Components

The PAPI library was originally developed to address the problem of accessing the processor hardware counters found on a diverse collection of modern microprocessors in a portable manner [1]. Other system components besides the processor, such as heterogeneous processors (GPUs), memory interface chips, network interface cards, and network switches, also have hardware that counts

various events related to system reliability and performance. Furthermore, other system health measurements, such as chip or board level temperature sensors, are available and useful to monitor in a portable manner. Unlike on-processor counters, the off-processor counters and sensors usually measure events in a system-wide rather than a process or thread-specific context. However, when an application has exclusive use of a machine partition, or runs in a single core of a multi-core node, it may be possible to interpret such events in the context of the application. Even with execution on multiple cores on a single node it may be possible to deconvolve the temperature or power signatures of separate threads to develop a coarse picture of single thread response. The current situation with off-processor counters is similar to the situation that existed with on-processor counters before PAPI. A number of different platform-specific interfaces exist, some of which are poorly documented or not documented at all.

Several software design issues became apparent in extending the PAPI interface for multiple measurement domains. The classic PAPI library consists of two internal layers: a large portable layer optimized for platform independence; and a smaller hardware specific layer, containing platform dependent code. By compiling and statically linking the independent layer with the hardware specific layer, an instance of the PAPI library could be produced for a specific operating system and hardware architecture. At compile time the hardware specific layer provided common data structure sizes and definitions to the independent layer, and at link time it satisfied unresolved function references across the layers. Since there was a one-to-one relationship between the independent layer and the hardware specific layer, initialization and shutdown logic was straightforward, and control and query routines could be directly implemented. In migrating to a multi-component model, this one-to-one relationship was replaced with a one-to-many coupling between the independent, or framework, layer and a collection of hardware specific components, requiring that previous code dependencies and assumptions be carefully identified and modified as necessary.

When linking multiple components into a common object library, each component exposes a subset of the same functionality to the framework layer. To avoid name-space collisions in the linker, the entry points of each component are modified to hide the function names, either by giving them names unique to the component, or by declaring them as static inside the component code. Each component contains an instance of a structure, or vector, with all the necessary information about opaque structure sizes, component specific initializations and function pointers for each of the functions that had been previously statically linked across the framework/component boundary. The only symbol that a component exposes to the framework at link time is this uniquely named component vector. All accesses to the component code occur through function pointers in this vector, and empty vector pointers fail gracefully, allowing components to be implemented with only a subset of the complete functionality. In this way, the framework can transparently manage initialization of and access to multiple components by iterating across a list of all available component structures. Our experiments have shown that the extra level of indirection introduced by

calls through a function pointer adds a small but generally negligible additional overhead to the call time, even in time-critical routines such as reading counter values. Timing tests were done on hardware including Intel Pentium4, Core2, and Nehalem, AMD Opteron and IBM POWER6 architectures. Over 1M iterations of a loop including 10 calls to empty subroutines the average execution time difference between direct and indirect calls was in the range of 6.9% for Nehalem to 46% for POWER6. In the context of real PAPI workloads on these same machines, a start/stop operation was slowed by between 0.13% and 1.36%, while a read of two counters was slowed by between 1.26% and 11.3%. Table 1 shows these results in greater detail.

**Table 1.** Costs of PAPI calls

| | Pentium4 | Core2 | Nehalem | Opteron | POWER6 |
|---|---|---|---|---|---|
| direct cycles/call | 13.8 | 8.4 | 5.8 | 9.6 | 106.3 |
| indirect cycles/call | 17.8 | 10.3 | 6.2 | 11 | 155.2 |
| % slowdown | 29.00% | 22.60% | 6.90% | 14.60% | 46.00% |
| PAPI start/stop slowdown | 0.66% | 0.52% | 0.13% | 0.39% | 1.36% |
| PAPI read 2 counters slowdown | 9.76% | 6.40% | 2.47% | 11.30% | 1.26% |

Countable events in PAPI are either preset events, defined uniformly across all architectures, or native events, unique to a specific component. To date preset events have only been defined for processor hardware counters, making all events on off-processor components native events.

### 2.1 Preset Events

Preset events can be defined as a single event native to a given CPU, or can be derived as a linear combination of native events, such as the sum or difference of two such events. More complex derived combinations of events can be expressed in reverse polish notation and computed at run-time by PAPI. The number of unique terms in these expressions is limited by the number of counters in the hardware. For many platforms the preset event definitions are provided in a comma separated values file, `papi_events.csv`, which can be modified by developers to explore novel or alternate definitions of preset events. Because not all preset events are implemented on all platforms, a utility called `papi_avail` is provided to examine the list of preset events on the platform of interest. A portion of the output for an Intel Nehalem (core i7) processor is shown below:

```
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 4.0.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel Core i7 (21)
CPU Revision             : 5.000000
CPUID Info               : Family: 6  Model: 26  Stepping: 5
CPU Megahertz            : 2926.000000
CPU Clock Megahertz      : 2926
Hdw Threads per core     : 1
Cores per Socket         : 4
```

```
NUMA Nodes              : 2
CPU's per Node          : 4
Total CPU's             : 8
Number Hardware Counters : 7
Max Multiplex Counters   : 32
-------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No    No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
...
PAPI_FP_OPS  0x80000066  Yes   Yes  Floating point operations
PAPI_SP_OPS  0x80000067  Yes   Yes  Floating point operations;
                                    optimized to count scaled single precision
                                    vector operations
PAPI_DP_OPS  0x80000068  Yes   Yes  Floating point operations;
                                    optimized to count scaled double precision
                                    vector operations
PAPI_VEC_SP  0x80000069  Yes   No   Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes   No   Double precision vector/SIMD instructions
------------------------------------------------------------------------
Of 107 possible events, 34 are available, of which 8 are derived.
```

### 2.2 Native Events

PAPI components contains tables of native event information allowing native events to be programmed in essentially the same way as a preset event. Each native event may have a number of attributes, called unit masks, that can act as filters on exactly what gets counted. These attributes can be appended to a native event name to tell PAPI exactly what to count. An example of a native event name with unit masks from the Intel Nehalem architecture is shown below:

`L2_DATA_RQSTS:DEMAND_M_STATE:DEMAND_I_STATE`

Attributes can be appended in any order and combination, and are separated by colon characters. Some components such as LM-SENSORS may have hierarchically defined native events. An example of such a hierarchy is shown below:

`LM_SENSORS.max1617-i2c-0-18.temp2.temp2_input`

In this case, levels of the hierarchy are separated by period characters. Complete listings of these and other native events can be obtained from a utility analogous to `papi_avail`, called `papi_native_avail`. A portion of the output of `papi_native_avail` for Nehalem configured with multiple components is shown below:

```
...
-------------------------------------------------------------------------------
0x40000032   L1I_OPPORTUNISTIC_HITS  | Opportunistic hits in streaming        |
-------------------------------------------------------------------------------
0x40000033   L2_DATA_RQSTS  | All L2 data requests                            |
  40000433   :ANY  | All L2 data requests                                     |
  40000833   :DEMAND_E_STATE  | L2 data demand loads in E state               |
  40001033   :DEMAND_I_STATE  | L2 data demand loads in I state (misses)      |
  40002033   :DEMAND_M_STATE  | L2 data demand loads in M state               |
  40004033   :DEMAND_MESI  | L2 data demand requests                          |
  40008033   :DEMAND_S_STATE  | L2 data demand loads in S state               |
  40010033   :PREFETCH_E_STATE  | L2 data prefetches in E state               |
```

```
  40020033   :PREFETCH_I_STATE  | L2 data prefetches in the I state (misses)   |
  40040033   :PREFETCH_M_STATE  | L2 data prefetches in M state                |
  40080033   :PREFETCH_MESI   | All L2 data prefetches                         |
  40100033   :PREFETCH_S_STATE  | L2 data prefetches in the S state            |
--------------------------------------------------------------------------------
0x40000034   L2_HW_PREFETCH   | Count L2 HW Prefetcher Activity                |
  40000434   :HIT  | Count L2 HW prefetcher detector hits                      |
  40000834   :ALLOC  | Count L2 HW prefetcher allocations                      |
  40001034   :DATA_TRIGGER  | Count L2 HW data prefetcher triggered            |
  40002034   :CODE_TRIGGER  | Count L2 HW code prefetcher triggered            |
  40004034   :DCA_TRIGGER  | Count L2 HW DCA prefetcher triggered              |
  40008034   :KICK_START  | Count L2 HW prefetcher kick started                |
--------------------------------------------------------------------------------
...
--------------------------------------------------------------------------------
0x44000000   ACPI_STAT  | kernel statistics                                    |
--------------------------------------------------------------------------------
0x44000001   ACPI_TEMP  | ACPI temperature                                     |
--------------------------------------------------------------------------------
0x48000000   LO_RX_PACKETS  | LO_RX_PACKETS                                    |
--------------------------------------------------------------------------------
0x48000001   LO_RX_ERRORS  | LO_RX_ERRORS                                      |
--------------------------------------------------------------------------------
...
--------------------------------------------------------------------------------
0x4c0000b3   LM_SENSORS.w83627hf-isa-0290.cpu0_vid.cpu0_vid                    |
--------------------------------------------------------------------------------
0x4c0000b4   LM_SENSORS.w83627hf-isa-0290.beep_enable.beep_enable              |
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Total events reported: 396
```

### 2.3 API Changes

An important consideration in extending a widely accepted interface such as PAPI is to make extensions in such a way as to preserve the original interface as much as possible for the sake of backward compatibility. Several entry points in the PAPI user API were augmented to support multiple components, and several new entry points were added to support new functionality.

By convention, an event to be counted is added to a collection of events in an *EventSet*, and *EventSets* are started, stopped, and read to produce event count values. Each *EventSet* in Component PAPI is bound to a specific component and can only contain events associated with that component. Multiple *EventSets* can be active simultaneously, as long as only one *EventSet* per component is invoked. The binding of *EventSet* and component can be done explicitly at the time it is created with a call to the new API:

```
PAPI_assign_eventset_component() - assign a component index to an existing
                                    but empty EventSet
```

Explicit binding allows a variety of attributes to be modified in an *EventSet* even before events are added to it. To preserve backward compatibility for legacy applications, binding to a specific component can also happen automatically when the first event is added to an *EventSet*.

Three entry points in the API allow access to settings within PAPI. These entry points are shown below:

```
PAPI_num_hwctrs() - return the number of hardware counters for the cpu
```

```
PAPI_get_opt()    - query the option settings of the PAPI library or
                    a specific event set
PAPI_set_domain() - set the default execution domain for new event sets
```

Component specific versions of these calls are:

```
PAPI_num_cmp_hwctrs() - return the number of hardware counters for the cpu
PAPI_get_cmp_opt()    - query the option settings of the PAPI library
                        or a specific event set
PAPI_set_cmp_domain() - set the default execution domain for new event sets
```

These modified callse have been implemented with an additional parameter to allow specification of given component within the call. Backward compatibility is preserved by assuming that the original calls are always bound to the original cpu component.

Finally two new calls were added to provide housekeeping functions. The first simply reports the current number of components, and the second returns a structure of information describing the component:

```
PAPI_num_components()
PAPI_get_component_info()
```

Neither of these calls are required. In this way legacy code instrumented with PAPI calls compiles and runs with no modification needed.

Example components have been implemented in the initial release of PAPI for ACPI temperature sensors, the Myrinet network counters, and the lm-sensors interface. An implementation of an Infiniband network component is under investigation, along with several other components for disk sub-systems such as Lustre.

### 2.4 The CPU Component

The CPU component is unique for several reasons. Historically it was the only component that existed in earlier versions of PAPI. Within Component PAPI one and only one CPU component must exist and occupy the first position in the array of components. This simplifies default behavior for legacy applications. In addition to providing access to the hardware counters on the main processor in the system, the CPU component also provides the operating system specific interface for things like interrupts and threading support, as well as high resolution time bases used by the PAPI Framework layer. The necessity for a unique CPU component has been identified as a restriction from the perspective of implementations that may not need or wish to monitor the CPU and also implementations that may contain heterogeneous CPUs. This is an open research issue in Component PAPI and mechanisms are under investigation to relax these restrictions.

### 2.5 Accessing the CPU Hardware Counters

CPU Hardware counter access is provided in a variety of ways on different systems. When PAPI was first released almost 10 years ago, there was significant

diversity in the operating systems and hardware of the Top500 list. AIX, Solaris, UNICOS and IRIX shared the list with a number of variants of Unix [8]. Linux systems made up a mere 3.6 percent of the list. Most of these systems had vendor provided support for counter access either built-in to the operating system, or available as a loadable driver. The exception was Linux, which had no support for hardware counter access. This is in sharp contrast to today [9], when nearly 90 percent of the systems run Linux or Linux variants.

Several options were available to access counters on Linux systems. One of the earliest was the `perfctr` patch [10] for x86 processors. `Perfctr` provided a low latency memory-mapped interface to virtualized 64-bit counters on a per process or per thread basis, ideal for PAPI's "first person" counting and sampling interface. With the introduction of Linux on the Itanium processor, the `perfmon` [5] interface was built-in to the kernel. When it became apparent that `perfctr` would not be accepted into the Linux kernel, `perfmon` was rewritten and generalized as `perfmon2` [11] to support a wide range of processors under Linux, including the IBM POWER series in addition to x86 and IA64 architectures. After a continuing effort over several years by the performance community to get `perfmon2` accepted into the Linux kernel, it too was rejected and supplanted by yet another abstraction of the hardware counters, first called `perf_counters` in kernel 2.6.31 and then `perf_events` [6] in kernel 2.6.32. The `perf_events` interface is young and maturing rapidly. It has the overwhelming advantage of being built-in to the kernel, requiring no patching on the part of system administrators. PAPI continues to support hardware counter access through `perfctr` wherever it is available. `Perfmon` access is available through the 2.6.30 kernel. In addition, PAPI also supports the `perf_events` interface.

### 2.6 The ACPI and MX Components

The ACPI component enables the PAPI-C library to access the ACPI temperature sensors, while the MX component allows monitoring of run-time characteristics of the Myrinet network communications. To demonstrate simultaneous monitoring of CPU metrics as well as temperature and data transfer, we collected data from the HPC Challenge suite. This suite is a set of scalable, computationally intensive benchmarks with different memory access patterns that examine the performance of HPC architectures [12]. For our experiments, we chose two global kernel benchmarks, High Performance Linpack (HPL) and FFT. The HPL kernel solves a linear system of equations and the FFT kernel computes a double precision complex one-dimensional discrete Fourier transform, which ensures two highly computationally intense test cases. We instrumented both benchmarks to gather total floating-point operations, temperature and packets sent and received through the Myrinet network. With Component PAPI, we were able to easily instrument the program by simply providing the desired event names in PAPI calls. We ran our experiments on a 65-node AMD Opteron cluster. Both benchmarks ran on eight nodes. We instrumented functions fft235 in FFT and pdgesvK2 in HPL, since profiling indicated that these were the most

computationally active routines, and gathered data for each iteration that called these functions.

The measurements for the FFT benchmark on two of the nodes are shown in Fig. 1. We can see the periodic nature of the computation and communication. The measured data for the second case study - the HPL benchmark - is depicted in Fig. 2 and shows a completely different computation and communication pattern. In both test cases, we are able to observe a difference in the temperature between the two nodes.

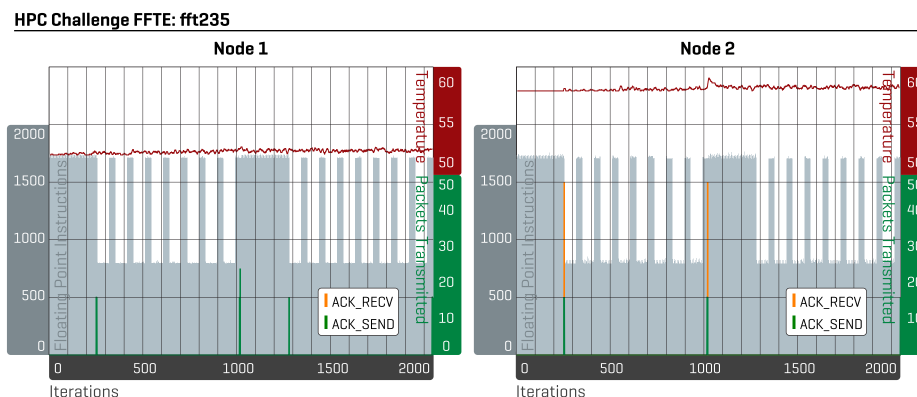


**Fig. 1.** FLOPS, temperature and communication monitoring using the CPU, ACPI and MX component of PAPI-C for an FFT benchmark running on an AMD Opteron cluster

## 2.7 The LM-SENSORS Component

The LM-SENSORS component enables the PAPI-C library to access all computer health monitoring sensors as exposed by the `lm_sensors` [13] library. The user is able to closely monitor the system's hardware health as an attempt to get more performance out of environmental conditions of the hardware. What features are available and what exactly can be monitored depends on the hardware setup.

We monitored three fan speeds as well as the CPU temperatures on a quad-core Intel Nehalem (core i7) machine using the LM-SENSORS component of PAPI-C. Multiple iterations of numeric operations are performed to heat up the compute cores. In total, 128 threads have been created and distributed over 8 compute cores and each of them executes the numeric code. The fan speeds as well as CPU temperatures are monitored every 10 seconds. Figure 3(a) shows

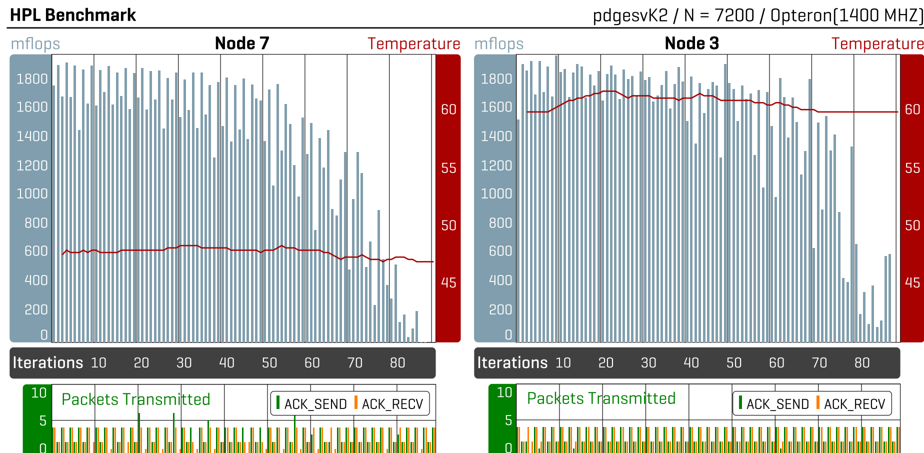


**Fig. 2.** FLOPS, temperature and communication monitoring using the CPU, ACPI and MX component of PAPI-C for an HPL benchmark running on an AMD Opteron cluster

the collected speed data of three fans while Fig. 3(b) depicts the temperature of the two quad-core CPUs. From those graphs, it is evident that the rotational speed of the fans responds to changes on the CPU temperature sensors. Note once more the difference in temperature between the two CPUs. We have seen similar correlation betweeen temperature and workload before in section 2.6 on a the Opteron architecture.

## 3 Multi-core Performance Measurement

With the arrival of the multi-core era for modern Petascale computing, more discussions are turning to the future implications of multi-core processors. The main focus in this section is the impact of shared resources of multi-core processors on the CPU component of PAPI-C which is described in 2.4. With the help of an application test case, we will discuss the difference between hardware performance data collection for on-core versus off-core resources. The current approach of collecting hardware performance counters shows serious limitations for off-core resources. However, measurement of performance counter data from shared resources is crucial in the analysis of scientific applications on multi-core processors due to the fact that this is where resource contention occurs. The key is to minimize the contention of shared resources such as caches, memory bandwidth, bus and other resources.

The multi-core transition in hardware design also reflects an impact on software development which remains a big challenge. To illustrate issues associated with the measurement of performance events for shared resources, we quantitatively evaluate the performance of the memory sub-system on Jaguar, the

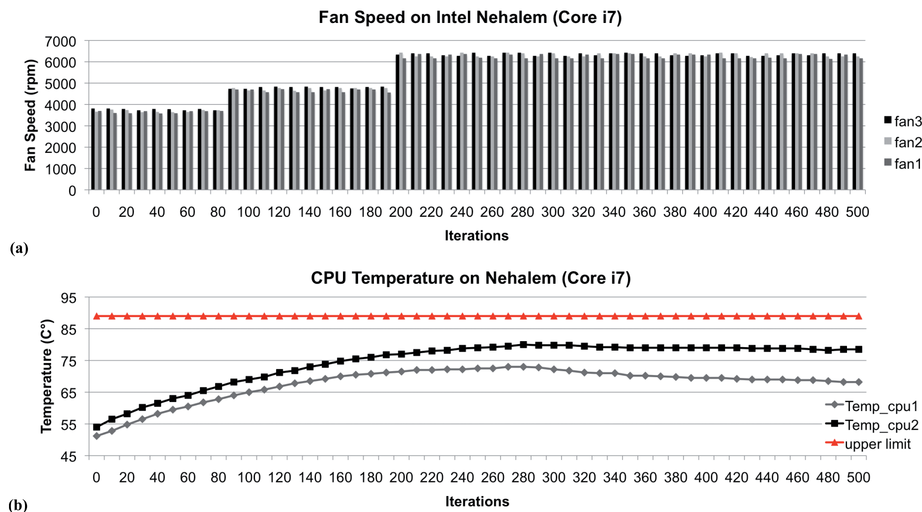


**Fig. 3.** (a) Fan speed monitoring; (b) CPU temperature monitoring - both metrics have been investigated on an Intel Nehalem (core i7) machine using the LM-SENSORS component of PAPI-C

fastest computer on the November 2009 Top500 list [14]. The Jaguar system at Oak Ridge National Laboratory (ORNL) has evolved rapidly over the last several years. When the work reported here was done, Jaguar was based on Cray XT4 hardware and utilized 7,832 quad-core AMD Opteron processors with a clock frequency of 2.1 GHz and 8 GBytes of memory (maintaining the per core memory at 2 GBytes). For more information on the Jaguar system and the quad-core AMD Opteron processor, the reader is referred to [15, 16].

The application test case is drawn from workload configurations that are expected to scale to large number of cores and that are representative of Petascale problem configurations. The massively parallel direct numerical simulation (DNS) solver (S3D) - developed at Sandia National Laboratories - solves the full compressible Navier-Stokes, total energy, species, and mass continuity equations coupled with detailed chemistry [17–19]. The application was run in SMP (one core per node) as well as VN mode (four cores per node) on Jaguar. Both test cases apply the same core count. The total execution time for runs using the two different modes shows a significant slowdown of 25% in VN mode (813 seconds) when compared to single-core mode (613.4 seconds). The unified L3 cache is shared between all four cores. We collected hardware performance events using the PAPI library that confirms our findings. L3 cache requests are measured and computed using the following PAPI native events:

```
L3 REQUESTS = READ REQUESTS TO L3 + L3 FILLS CAUSED BY L2 EVICTION
```

*Note: In VNM all L3 cache measurements have been divided by 4 (4 cores per node)*

Figure 4 (a) depicts the number of L3 cache misses and requests when using four cores versus one core per node for the 13 most expensive functions of the S3D application. It appears that the performance degradation in VN mode is due to the L3 cache behavior. In VN mode we see roughly twice as many L3 cache requests and misses compared to SMP mode. It is not surprising that L3 cache misses increase with VN mode since if every thread is operating on different data, then one thread could easily evict the data for another thread if the sum of the four working threads is greater than the size of the L3 cache. However, the increase in L3 requests is rather questionable. The L3 cache serves as a victim cache for L2. In other words, a datum evicted from L2 (the victim) is deposited in L3. If requested data is not in L2 cache then the L3 cache is checked which results in an L3 request. While the L3 cache is shared between all four cores, the L2 cache remains private. Based on this workflow, it is not clear why the number of L3 requests increases so dramatically when using all four cores per node. As verification we measure the L2 cache misses in SMP and VN mode and Fig. 4 (b) presents the comparison. It clearly shows that the number of L2 cache misses does not increase when all four cores are used compared to SMP mode. All the more, the question persists as to where the double L3 cache requests come from when VN mode is used. It is important to note, the policy on the Jaguar system defines that by default a task - independent of process or thread - is not allowed to migrate to a CPU core within a socket or to any CPU core on either socket [20]. For the S3D test case, we applied this default configuration which pins a task to a specific CPU core.

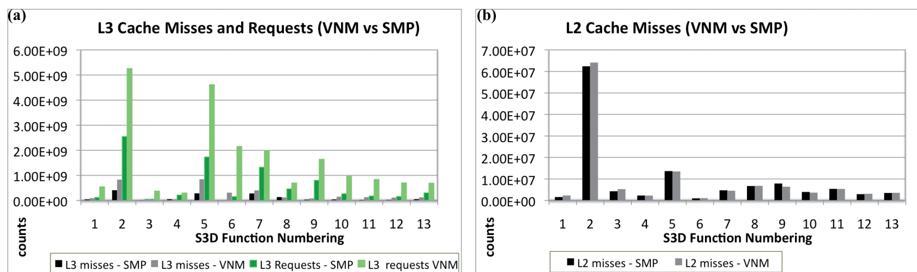


**Fig. 4.** (a) L3 cache misses and requests (mean); (b) L2 cache misses (mean)

### 3.1 Various Multi-core Designs

Recent investigations and discussions have suggested that the high L3 cache request rate in S3D may be an artifact of the measurement process. Current Opteron hardware is not designed for first-person counting of events involving shared resources [21]. The L3 events in AMD Opteron quad-core processors are not monitored in four independent sets of hardware performance registers but

in a single set of registers not associated with a specific core (often referred to as "shadow" registers). Each core has four independent counter registers which are used for most performance events. When an L3 event is programmed into one of these counters on one of these cores, it gets copied by hardware to the shadow register. Thus, only the last event to be programmed into any core is the one actually measured by all cores. When several cores try to share a shadow register, the results are not clearly defined. Performance counter measurement at the process or thread level relies on the assumption that counter resources can be isolated to a single thread of execution. That assumption is generally no longer true for resources shared between cores - like the L3 cache in AMD quad-core processors.

This problem is not isolated just to AMD Opteron processors. Early Intel dual-core processors addressed the issue of measuring shared resources by providing SELF and BOTH modifiers on events involving shared caches or other resources. This allowed each core to independently monitor the event stream for a shared resource and to either collect only counts for its activity or for all activities involving that resource. However, with the introduction of the Nehalem (core i7) architecture, Intel, too, moved measurement of chip level shared resources off the cores and onto the chips. The Nehalem architecture includes eight "Uncore" counters [22] that are shared among all the cores of the chip. There is presently no mechanism for a given core to reserve counter resources from the Uncore. These events can be monitored by the `perfmon2` [5] patch, but only in system-wide counting mode. Thus these counter measurements cannot be performed with a first-person measurement paradigm such as PAPI's, and cannot be intermixed with per process measurements of other events. The built-in `perf_events` [6] module in the Linux kernel has no support for Uncore counters as of the 2.6.32 kernel release.

A final example of the multi-core problem of measuring activities on shared resource is IBM's Blue Gene series. Blue Gene/L is a dual-core processor and Blue Gene/P is a quad-core processor. In both cases hardware counters are implemented in the UPC, a Universal Performance Counter module that is completely external to any core. In Blue Gene/P for example, the UPC contains 256 independent hardware counters [23]. Events on each core can be measured independently, but the core must be specified in the event identifier. This can create great difficulty for code that in general does not know or care on which core it is running. Further, these counters can only be programmed to measure events on either core 0 and 1, or core 2 and 3, but not on a mixture of all four cores at once.

As the above examples illustrate, hardware vendors are searching for ways to provide access to performance events on shared resources. There is presently no standard mechanism that provides performance information in a way that is useful for software tuning. New methods need to be developed to appropriately collect and interpret hardware performance counter information collected from such multi-core systems with interesting shared resources. PAPI research is underway to explore these issues.

## 4 Future Directions

With the release of PAPI-C, the stage is set for a wide range of development directions. Our major goals with the first release were stability and compatibility. As with any research and development effort there are always open issues to be explored. Here are some of the issue under investigation with Component PAPI:

- **Event Naming:** PAPI presently expresses all events as 32-bit event codes. With the richness of current events and attributes and modifiers, we find this too restrictive, and will be migrating to a model in which all events are referenced by name.
- **Data Types:** PAPI supports returned data values expressed as unsigned 64-bit integers. This is appropriate for counting events, but may not be as appropriate for expressing other values. We are exploring ways to encode and specify other 64-bit data formats including: signed integer, IEEE double precision, fixed point, and integer ratios.
- **Dynamic Configurability:** The current mechanism for adding new components is workable, but not well suited to introducing new components between releases of the PAPI Framework. Methods are needed for an automated discovery process for components, both at build time and at execution time.
- **Synchronization:** Components can report values with widely different time scales and remote measurements may exhibit significant skew and drift in time from local measurements. Mechanisms need to be developed to accomodate these artifacts.
- **Component Management:** To encourage users and third parties to become component contributors, efforts will be invested in documenting the component development process and in managing 3rd party components.

At a recent brainstorming session by the PAPI developers, a number of future directions for the PAPI project were identified. In a somewhat whimsical fashion, and building on the idea of the PAPI-C name, several new letters for the PAPI "alphabet soup" were put forth:

- **PAPI-M: Multi-core.** The issue of how to measure shared resource performance on a variety of multi-core architectures remains unresolved. This may require more kernel development than PAPI development, but is an important issue that should be addressed.
- **PAPI-G: GPUs.** GPGPUs and other heterogenous compute elements will be an increasingly important part of our computing eco-system as we move from Petascale to Exascale. They present radically different sorts of performance information to the user and provide a challenging opportunity for performance presentation.
- **PAPI-V: Virtual.** With access to performance hardware now part of the Linux kernel, it becomes possible to introduce this information into the hypervisors that comprise virtual, or cloud, computing space. With support in the hypervisors, it becomes possible to consider what it means to measure hardware performance in the cloud.

- **PAPI-N: Networks.** As core counts rise exponentially on the march to Exascale, communication becomes even more dominant over computation as a determinant of execution time. PAPI-C components can be developed either in the open source community or by vendors to monitor hardware characteristics of either open network standards such as Infiniband or proprietary hardware such as Cray's SeaStar or Gemini network chips.
- **PAPI-D: Disks.** Several users of PAPI have suggested and begun work on the development of PAPI Components to measure remote Disk storage activities for file systems like Lustre. Such information could prove useful in managing and measuring the impact of storage operations on execution performance.
- **PAPI-H: Health.** System health measurements are often done out-of-band from compute activities. PAPI-C components may be developed to run on system nodes in parallel with jobs on compute nodes to assess the impact of application activities on temperature or power consumption, or to warn of impending resource failure and the need for remedial action.

## 5 Conclusion

For most of the past decade, PAPI has been the de-facto choice to provide the tool designer and application engineer with a consistent interface for accessing hardware performance counters on a wide range of computer architectures. PAPI has ridden the evolutionary wave of processor development as clock rates, pipeline depth and instruction level parallelism increased through the decade. That smooth evolution has recently ended with the flattening of clock rates, the introduction of multi-core architectures, the adoption of heterogeneous computing approaches and the need for more careful monitoring of system health required for fault tolerance and resiliency in the Petascale domain of hundreds of thousands of processors. We are now in a period of punctuated equilibrium where the paradigms of the recent past are being swept away by a tidal wave of changes at a number of levels.

The development of Component PAPI for the simultaneous monitoring of multiple measurement domains positions this library to remain as a central tool in the acquisition of performance data across a spectrum of architectures and activities. This extension has been done in such a way as to cause minimal disruption to the current user base while providing flexible opportunities to gain new insights into application and system performance.

### Acknowledgements

This research was sponsored in part by the Office of Mathematical, Information, and Computational Sciences of the Office of Science (OoS), U.S. Department of Energy (DoE), under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This work used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of

Science of the Department of Energy under Contract DE-AC05-00OR22725. These resources were made available via the Performance Evaluation and Analysis Consortium End Station, a Department of Energy INCITE project.

This work was also supported in part by the U.S. Department of Energy Office of Science under contract DE-FC02-06ER25761; by the National Science Foundation under Grant No. 0910899 [†] as well as Software Development for Cyberinfrastructure (SDCI) Grant No. NSF OCI-0722072 Subcontract No. 207401; and by the Department of Defense, using resources at the Extreme Scale Systems Center.

## References

1. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. International Journal of High-Performance Computing Applications, , Vol. 14, No. 3, pp. 189-204 (2000)
2. Cameron, K.W., Ge, R., and Feng, X.: High-performance, power-aware distributed computing for scientific applications. Computer, 38(11):4047 (2005)
3. Feng, W.C.: The importance of being low power in high performance computing. CTWatch Quarterly, 1(3), August (2005)
4. Freeh, V.W., Lowenthal, D.K., Pan, F., Kappiah, N.: Using multiple energy gears in MPI programs on a power-scalable cluster. In Principles and Practices of Parallel Programming (PPOPP), June (2005)
5. `Perfmon2` Sourceforge Project Page: `http://perfmon2.sourceforge.net`
6. Molnar, I.: Performance Counters for Linux, v8. `http://lwn.net/Articles/336542`
7. Moore, S.: A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware. ICCS 2002, Amsterdam, April (2002)
8. Operating System share, November 1999: `http://www.top500.org/charts/list/14/os`
9. Operating System share, November 2009: `http://www.top500.org/charts/list/34/os`
10. Pettersson, M.: Linux x86 Performance-Monitoring Counters Driver. `http://www.csd.uu.se/~mikpe/linux/perfctr`
11. Jarp, S., Jurga, R., Nowak, A.: Perfmon2: A leap forward in Performance Monitoring. Journal of Physics: Conference Series 119, 042017 (2008)
12. Luszczek, P., Dongarra, J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D.: Introduction to the hpc challenge benchmark suite. Technical report, March (2005)
13. Hardware Monitoring by `lm_sensors`: `http://www.lm-sensors.org/`
14. Top500 list: `http://www.top500.org`
15. NCCS.gov computing resources documentation: `http://www.nccs.gov/computing-resources/jaguar`
16. Software Optimization Guide for AMD Family 10h Processors, Pub. no. 40546 (2008)
17. Chen, J. H., Hawkes, E. R., et al.: Direct numerical simulation of ignition front propagation in a constant volume with temperature inhomogeneities I. fundamental analysis and diagnostics. Combustion and flame, 145, pp. 128-144 (2006)

[†] This acknowledgment was inadvertently omitted from the published version "Tools for HPC 2009", Springer Berlin / Heidelberg 2010, pp. 157-173.

18. Sankaran, R., Hawkes, E. R., et al.: Structure of a spatially developing turbulent lean methane-air Bunsen flame. Proceedings of the combustion institute 31, pp. 1291-1298 (2007)
19. Hawkes, E. R., Sankaran, R., et al.: Scalar mixing in direct numerical simulations of temporally evolving nonpremixed plane jet flames with skeletal CO-H2 kinetics. Proceedings of the combustion institute 31, pp. 1633-1640 (2007)
20. Cray XT Programming Environment User's Guide (Version 2.2). S-2396-22, July (2009)
21. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 10h Processors (particularly Section 3.12.). Vol. 31116 Rev 3.34, September (2009)
22. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide (Particularly Chapter 19.17.2 Performance Monitoring Facility in the Uncore). Part 2 Order Number: 253669-031US, June (2009)
23. Walkup, B.: Blue Gene/P Universal Performance Counters. http://www.nccs.gov/wp-content/training/2008_bluegene/BobWalkup_BGP_UPC.pdf